**FIG. 1**

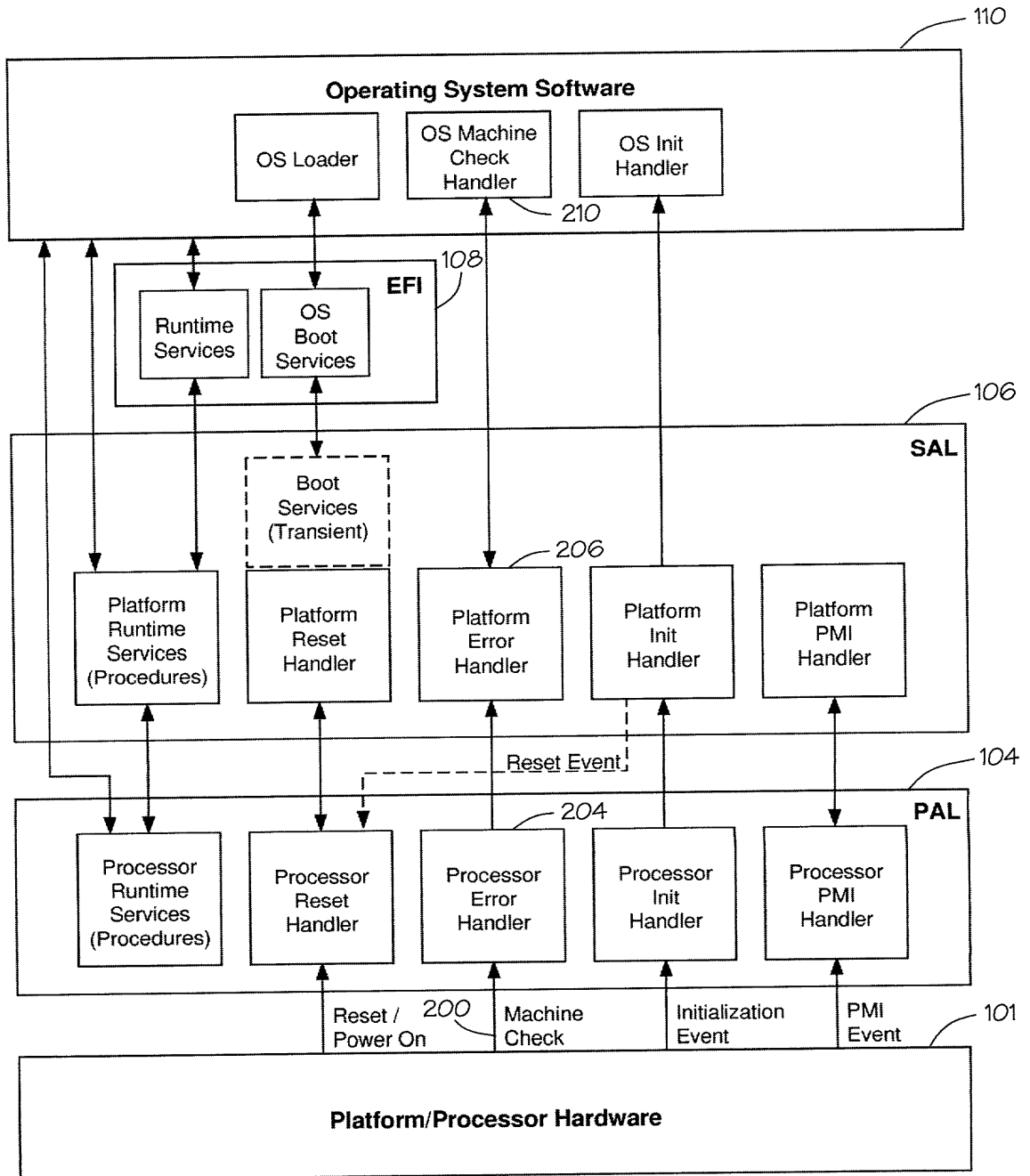


FIG. 2

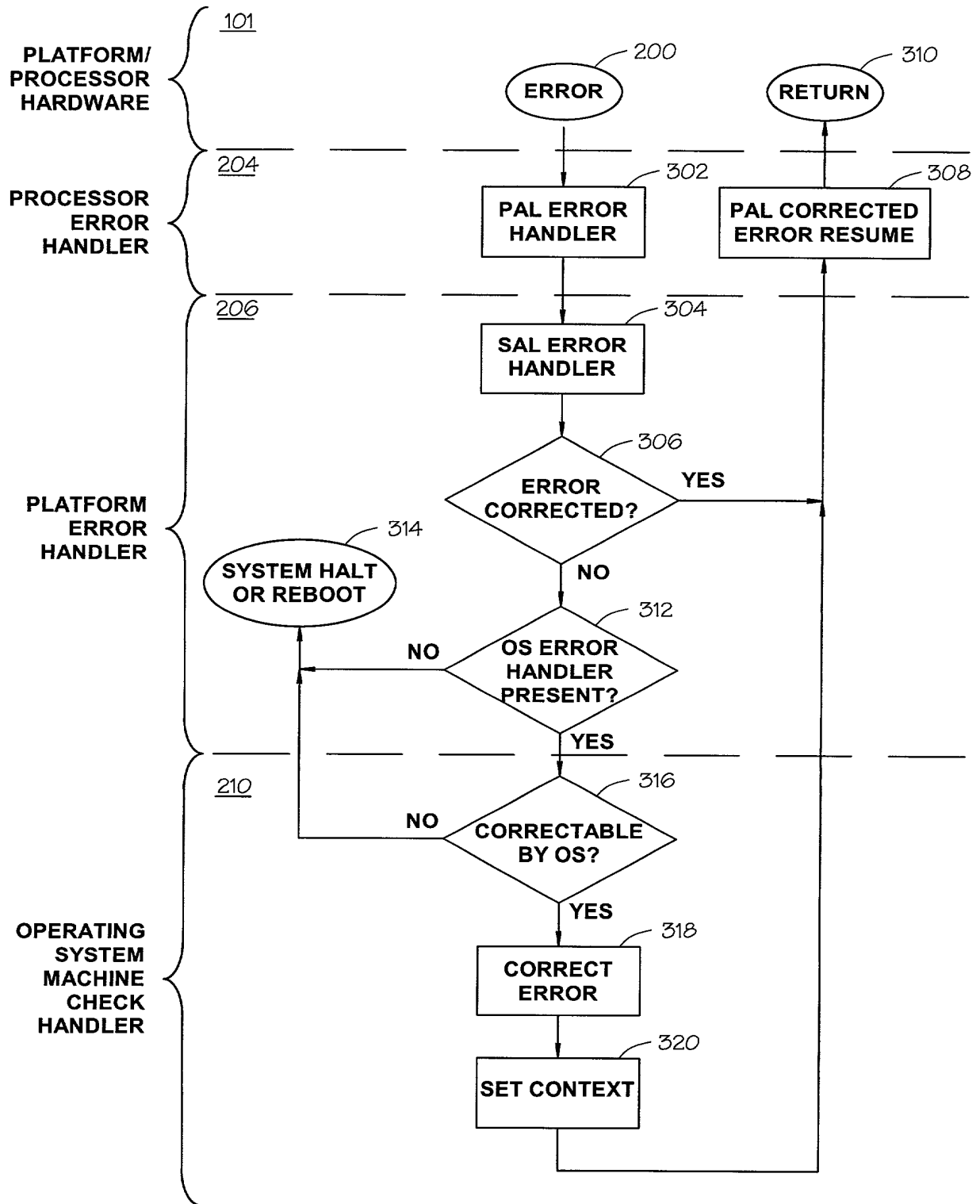


FIG. 3

```

/*=====*/
/* Definitions - These are provided to attempt to make the pseudo */
/* code easier to read and are not meant to be real */
/* definitions that can be used. */
/*=====*/

/* Processor State Parameter is located in PSP=r18 at hand off from */
/* SAL to the OS_MCA handler. */

/* Processor State Parameter bit field definitions */
define TLB_Error = ProcessorStatParameter[60]

/* SAL Record Header Error Log Definitions */

#define Record_ID_Offset = 0
#define Err_Severity_Offset = 10
#define Recoverable = 0
#define Fatal = 1
#define Corrected = 2
#define Record_Length_Offset = 12
#define Record_Header_Length = 24

/* SAL Section Header Error Log Definitions */

#define GUID_Offset = 0
#define Section_Length_Offset = 20
#define Processor_GUID = E429FAF1-3CB7-11D4-BCA70080C73C8881
#define Section_Header_Length = 24

/* SAL Processor Error Record Definitions */

#define Valdiation_Bit_Structure
    Proc_Error_Map_Valid = bit 0
    Cache_Check_Valid = bits [7:4]
    TLB_Check_Valid = bits [11:8]
    Bus_Check_Valid = bits [15:12]
    Reg_File_Check_Valid = bits [19:16]
    MS_Check_Valid = bits [23:20]

#define Error_Validation_Bit_Length = 8
#define Check_Info_Valid_Bit = bit 0
#define Target_Address_Valid_Bit = bit 3
#define Precise_IP_Valid_Bit = bit 4

#define Check_Info_Offset = 0
#define Target_Address_Offset = 24
#define Precise_IP_Offset = 32

/* Cache Check Info Bit definitions */

#define PrecisePrivLevel = bits [57:56]
#define PrecisePrivLevel_Valid = bits 58

```

FIG. 4A

```

/*=====BEGIN=====*/
/* OS Machine Check Initialization */
/*=====*/
OS_MCA_Initialization( )
{
/* this code is executed once by OS during boot Register OS_MCA */
/* Interrupt parameters by calling SAL_MC_SETPARAMS */

    Install_OS_Rendez_Interrupt_Handler
    Install_OS_Rendez_WakeUp_Interrupt_Handler /* ISR clean up wrapper */
    Register_Rendez_Interrupt_Type&Vector;
    Register_WakeUpInterrupt_Type&Vector;
    Register_CorrectedPlatformErrorInterrupt_Vector;
    Initialize_CMC_Vector_Masking;

/* Register OS_MCA Entry Point parameters by calling SAL_SET_VECTORS */

    Register_OS_MCA_EntryPoint;
    Register_OS_INIT_EntryPoint;
}
/*=====END=====*/

/*=====BEGIN=====*/
/* OS Machine Check Rendez Interrupt Handler */
/*=====*/
OS_Rendez_Interrupt_Handler( )
{
    /* go to spinloop */
    Mask_All_Interrupts;
    Call SAL_MC_RENDEZ( );

    /* clean-up after wakeup from exit */
    Enable_All_Interrupts;

    /* return from interruption */
    return;
}
/*=====END=====*/

/*=====BEGIN=====*/
/* OS Corrected Error Interrupt Handler (processor and platform) */
/*=====*/
OS_Corrected_Error_Interrupt_Handler( )
{
/* handler for corrected machine check intr.*/
/* get error log */
if(ProcessorCorrectedError)
    Sal_Get_State_Info( processor);
else
    Sal_Get_State_Info(platform);
}

```

FIG. 4B

```

/* If saving of the error record is to disk or the OS event log, */
/* then this is core OS functionality. */

/* Save log of MCA */
Save_Error_Log( );

/* now we can clear the errors */
if(ProcessorCorrectedError)
    Call Sal_Clear_State_Info(processor);
else
    Call Sal_Clear_State_Info(platform);

/* return from interruption */
return;
}
/*=====END=====*/

/*-----BEGIN-----*/
/* OS Core Machine Check Handler */
/*-----*/
OS_MCA_Handler( )
{
/* handler for uncorrected machine check event */
    Save_Processor_State();

    if(ErrorType!=Processor TLB)
        SwitchToVirtualMode();
    else
        StayInPhysicalMode();

/* Assuming that the OS can call SAL in physical mode to get info */
    SAL_GET_STATE_INFO(MCA);

/* check for error */
    if(ErrorType==processor)
    {
        if(ErrorType=processor TLB)
            // cannot do much;
            // reset the system and get the error record at reboot
            SystemReset() or ReturnToSAL(failure);
        else
            ErrorCorrectedStatus=OsProcessorMca();
    }
    If(ErrorType==Platform)
        ErrorCorrectedStatus|=OsPlatformMca();

/* If the error is not corrected, OS may want to reboot the machine */
/* and can do it by returning to SAL with a failure return result. */

    If(ErrorCorrectedStatus==failure)
        branch=ReturnToSAL_CHECK

/* Errors are corrected, so try to wake up processors which are */
/* in Rendezvous. */

```

FIG. 4C

```

/* completed error handling */
If(ErrorCorrectedStatus=success && InRendezvous()==true)
    WakeUpApplicationProcessorsFromRendezvous();

/* If saving of the error record is to disk or the OS event log, */
/* then this is core OS functionality. */

/* as a last thing */
Save_Error_Log();

/* This is a very important step, as this clears the error record */
/* and also indicates the end of machine check handling by the OS. */
/* SAL uses this to clear any state information it may have related */
/* to which processors are in the MCA and any State of earlier */
/* rendezvous. */

Call Sal_Clear_State_Info(MCA);

ReturnToSAL::
    /* return from interruption */
    SwitchToPhysicalMode();
    Restore_Processor_State();

    /* return to SAL CHECK, SAL would do a reset if OS fails to correct */
    return(ErrorCorrectedStatus)
}
/*=====END=====*/

/*=====BEGIN=====*/
/* Os Platform Machine Check Handler */
/*=====*/
OsPlatformMca()
{
    ErrorCorrected=True;

    /* check if the error is corrected by PAL or SAL */
    If(ErrorRecord.Severity==not corrected)
        /* call sub-routine to try and correct the Platform MCA */
        ErrorCorrected=Correctable_Platform_MCA(platform_error_type);

    Return(ErrorCorrectedStatus);
}
/*=====END=====*/

/*=====BEGIN=====*/
/* OS Processor Machine Check Handler */
/*=====*/
OsProcessorMca( )
{
    ErrorCorrected=True;

    /* check if the error is corrected by Firmware */
    If(ErrorRecord.Severity==not corrected)
        ErrorCorrectedStatus=TryProcessorErrorCorrection( );

    Return(ErrorCorrectedStatus);
}
/*=====END=====*/

```

FIG. 4D

```

/*=====BEGIN=====*/
/* Try Individual Processor Error Correction */
/*=====*/

/* Now the OS has the data logs. Start parsing the log retrieved from */
/* SAL. The sub-routine Read_OS_Error_Log will read data from the error */
/* log copied from SAL. An offset is passed to identify the data being */
/* read and the base pointer is assumed to be known by the */
/* Read_OS_Error_Log sub-routine just to simplify the pseudo-code. */

TryProcessorErrorCorrection( )
{
    /* extract appropriate fields from the record header */
    Record_ID = Read_OS_Error_Log(Record_ID_Offset);
    Severity = Read_OS_Error_Log(Err_Severity_Offset);

    /* It is unlikely that the OS can write to persistent storage in */
    /* physical mode. If it is possible, the OS should do so. If it is not, */
    /* the SAL firmware should still have a copy of the error log stored */
    /* to NVRAM that will be persistent across resets. */

    if (Severity == Fatal)
        SystemReset() or return(failure);
    if (Severity == Corrected)
        return(ErrorCorrectedStatus=True);

    /* These errors may be recoverable by the OS depending on the OS */
    /* capability and the information logged by the processor. Call the */
    /* sub-routine, OS_MCA_Recovery_Code and on return set up a min-state */
    /* save area to return to a context of choice. The pal_mc_resume done */
    /* through SAL allows the OS to turn on address translations and enable */
    /* machine check aborts to be able to handle nested MCAs. */

    if (Severity == Recoverable)
    {
        ErrorCorrectedStatus=OS_MCA_Recovery();
        Set_Up_A_Min_State_For_OS_MCA_Recovery(my_minstate);
    }
    return(ErrorCorrectedStatus);
} /* End of TryProcessorErrorCorrection Handler */
/*=====END=====*/

/*=====BEGIN=====*/
/* OS MCA Recovery Code */
/*=====*/

/* At this point the OS is running with address translations enabled. */
/* This is needed otherwise the OS would not be able to access all of */
/* its data structures needed to analyze if the error is recoverable */
/* or not. There is a chance another MCA may come during recovery due */
/* to this fact, but running in physical mode for the OS is difficult */
/* to do. */

OS_MCA_Recovery( )
{
    /* Set up by default that the errors are not corrected */
    CorrectedErrorStatus = CorrectedCacheErr = CorrectedTlbErr =
    CorrectedBusErr = CorrectedRegFileErr = CorrectedUarchErr = 0;

```

FIG. 4E


```

/* Start parsing the error log */
RecordLength = Read_OS_Error_Log(Record_Length_Offset);
Section_Header_Offset = OS_Error_Log_Pointer + Record_Header_Length;

/* Find the processor error log data */
Processor_Error_Log_Found = 0;

/* traverse the error record structure to find processor section */
while (Processor_Error_Log_Found == 0)
{
    SectionGUID = Read_OS_Error_Log(Section_Header_Offset +
    GUID_Offset);
    SectionLength = Read_OS_Error_Log(Section_Header_Offset +
    Section_Length_Offset);

    if (SectionGUID == Processor_GUID)
        Processor_Error_Log_Found = 1;

    Section_Body_Pointer = Section_Header_Offset +
    Section_Header_Length;
    Section_Header_Offset = Section_Header_Offset + SectionLength;

    if (Section_Header_Offset >= RecordLength)
        InternalError(); /* Expecting a processor log */
}

/* Start parsing the processor error log. Section_Body_Pointer was set */
/* up to point to the first offset of the processor error log in the */
/* while loop above. Check the valid bits to see which part of the */
/* structure has valid info. The Read_OS_Error_Log sub-routine is */
/* assumed to know the initial pointer and just an offset is passed. */
/* This was done to allow the pseudo-code to be more readable. */

Proc_Valid_Bits = Read_OS_Error_Log(Section_Body_Pointer);
Section_Body_Pointer = Section_Body_Pointer + Validation_Bit_Length;

/* Read the Processor Error Map if the valid bit is set. */
if (Proc_Valid_Bits[Proc_Error_Map_Valid] == 1)
    Proc_Error_Map = Read_OS_Error_Log(Section_Body_Pointer);

/* Extract how many errors are valid in the error log and determine
which type */
Cache_Check_Errs = Proc_Valid_Bits[Cache_Check_Valid];
TLB_Check_Errs = Proc_Valid_Bits[TLB_Check_Valid];
Bus_Check_Errs = Proc_Valid_Bits[Bus_Check_Valid];
Reg_File_Errs = Proc_Valid_Bits[Reg_File_Check_Valid];
Uarch_Errs = Proc_Valid_Bits[MS_Check_Valid];

/* These sub-routines will return an indication of if the error can be
corrected by killing the affected processes. */
if (Cache_Check_Errs != 0)
{
    /* Check to see if one or multiple cache errors occurred */
    if (Cache_Check_Errs == 1)
        CorrectedCacheErr =
            Handle_Single_Cache_Error(Section_Body_Pointer);
    else
        CorrectedCacheErr =
            Handle_Multiple_Cache_Errors(Section_Body_Pointer);
}

```

FIG. 4F

```

if (TLB_Check_Errs != 0)
{
    /* Check to see if one or multiple TLB errors occurred */
    if (TLB_Check_Errs == 1)
        CorrectedTlbErr = Handle_Single_TLB_Error(Section_Body_Pointer);
    else
        CorrectedTlbErr =
            Handle_Multiple_TLB_Errors(Section_Body_Pointer);
}

if (Bus_Check_Errs != 0)
{
    /* Check to see if one or multiple Bus errors occurred */
    if (Bus_Check_Errs == 1)
        CorrectedBusErr =
            Handle_Single_Bus_Error(Section_Body_Pointer);
    else
        CorrectedBusErr =
            Handle_Multiple_Bus_Errors(Section_Body_Pointer);
}

if (Reg_File_Errs != 0)
{
    /* Check to see if one or multiple Register file errors occurred */
    if (Reg_File_Errs == 1)
        CorrectedRegFileErr =
            Handle_Single_Reg_File_Error(Section_Body_Pointer);
    else
        CorrectedRegFileErr =
            Handle_Multiple_Reg_File_Errors(Section_Body_Pointer);
}

if (Uarch_Errs != 0)
{
    /* Check to see if one or multiple uarch file errors occurred */
    if (Uarch_Errs == 1)
        CorrectedUarch_Err =
            Handle_Single_Uarch_Error(Section_Body_Pointer);
    else
        CorrectedUarch_Err =
            Handle_Multiple_Uarch_Errors(Section_Body_Pointer);
}

CorrectedErrorStatus = CorrectedCacheErr | CorrectedTlbErr |
    CorrectedBusErr | CorrectedRegFileErr |
    CorrectedUarch_Err;

return(CorrectedErrorStatus);
} /* end OS_MCA_Recovery_Code */
/*=====END=====*/

```

FIG. 4G

```

/*=====BEGIN=====*/
/* Single Cache Error Recovery Code */
/*=====*/
Handle_Single_Cache_Error
{
    /* Initialize variables to a known value */
    Cache_Check_Info = Target_Address_Length = Precise_IP_Info = -1;
    Cache_Check_Valid_Bits = Read_OS_Error_Log(Section_Body_Pointer);
    Section_Body_Pointer = Section_Body_Pointer
    +Error_Validation_Bit_Length;

    if (Check_Info_Valid_Bit == 1)
        Cache_Check_Info = Read_OS_Error_Log(Section_Body_Pointer +
        Check_Info_Offset);

    if (Target_Address_Valid_Bit == 1)
        Target_Address_Info = Read_OS_Error_Log(Section_Body_Pointer +
        Target_Address_Offset);

    if (Precise_IP_Valid_Bit == 1)
        Precise_IP_Info = Read_OS_Error_Log(Section_Body_Pointer +
        Precise_IP_Offset);

    /* Determine if the Target Address was captured by the processor or */
    /* not. If it was, determine if it points to global memory, shared */
    /* memory or if it is private. If it points to a global memory */
    /* structure, then a system reboot is necessary. If it is shared */
    /* or private it may be recoverable. */

    // if no target physical address is captured, then we have to reboot
    if(Target_Physical_Address_TarId=Not_Valid)
        SystemReset() or return(failure);

    // target physical address is captured, check with OS if this is
    // global address page
    if(OsIsTargetAddressGlobal(TarId))
        SystemReset() or return(failure) // in global page, it is bad news

    /* Now we know that the target address does not point to shared */
    /* memory. Check to see if a precise instruction pointer was captured. */
    /* If it was then check to see if it is a user or kernel IP. If we */
    /* have the precise IP map to the processes and kill it, else we have */
    /* to kill processes based on target address. */

    // so far so good, TardID is in local page: Do we have precise IP?
    if(PreciseIP==true)
    {
        // yes, precise IP is captured, so take this branch
        if(OsIsIpInKernelSpace(IP))
        {
            // IP in kernel space
            KernelSpaceIpFlag=1;
            if(OsIsProcessCritical(IP,0)==true)
                SystemReset();
            else

```

FIG. 4H

```

        {
            // kill all non-critical OS processes at IP
            OsKillAllProcesses(IP,0);
            return(success);
        }
    }
else
{
    // IP is in user space
    UserSpaceIpFlag=1;
    // kill all shared user processes
    OsKillAllProcesses(IP,0);
    return(success);
}
}
else

/* We do not have precise IP, so try to map the Target physical */
/* address to a processes. If the target address points to shared */
/* data, then all sharing processes need to be killed. If the */
/* target address points to a private page (global has been checked */
/* above) then just kill the offending process. */

{
    // Try and map Target Physical Address to a process data area
    if(PreviledgeLevel==Valid) //check if previledge level is valid
    {
        // ipl=Instruction Priviledge level
        if(ipl==user_level) // at user_level
        {
            // this is user priveledge level
            OsKillAllProcesses(0,TarId);
            return(rv);
        }
        else // kernel level
        {
            /* If the OS has a way to determine if the IP is in a critical part */
            /* of the kernal this can determine if the kernal process can be */
            /* killed or not. If the OS always puts critical kernal code in a */
            /* certain IP range, this could be a way it could determine. */

            // this is kernel priviledge level
            if(OsIsProcessCritical(0,TarId))
            {
                // OS critical process error, all bets are off...
                SystemReset() or return(failure);

                // good, can kill all non-critical processes using TardId
                OsKillAllProcesses(0,TarId);
                return(success);
            }
        }
    }
else
    // sorry, don't have privilege level information, all bets
    are off...
    SystemReset() or return(failure);
}

return(succcess);
}
/*=====END=====*/

```

FIG. 4I